Homework 5: Aspect-Oriented Programming and AspectJ

Due: Problems 1–7, Thursday, November 18, 2004; problems 8–15, Tuesday, November 30, 2004; problems 16–19, Thursday, December 2, 2004.

This homework should all be done individually. Its purpose is to help you explore aspect-oriented programming and the design of AspectJ.

Don't hesitate to contact the staff if you are not clear about what to do.

See the syllabus for readings for this homework, which includes the book by Laddad, the article by Kiczales, *et al.* we passed out, as well as the material on the `eclipse.org/aspectj` website, especially the *Programmer's Guide*. The course web page has material on running Java, and the AspectJ compiler as well as the Eclipse IDE are installed already on the department Linux machines. The eclipse IDE is run by using the command `eclipse`. The command-line AspectJ compiler is called `ajc`; you can also get a copy of either or both of these for your home machine from `eclipse.org`.

1. (suggested practice) To begin copy the directory

   ```
   /home/course/cs541/public/homework/aspectjhw
   ```

   and its contents to your home directory. (On unix, use `cp -r`.) To work with the following problems, you must have the jar files for JUnit and AspectJ's runtime in your `CLASSPATH`, as well as the directory that is just above `aspectjhw` (i.e., your home directory if you copied it there directly). For example, you can have on Linux, with the bash shell:

   ```
   CLASSPATH="$HOME:/opt/junit/junit.jar:/opt/aspectj/lib/aspectjrt.jar:."
   export CLASSPATH
   ```

   (You can put this in your .bashrc file if you wish, but you may have other Java projects that need other `CLASSPATH` settings.)

   If you use tcsh or csh as your shell (again on Linux), you should instead of the above execute something like:

   ```
   setenv CLASSPATH "$HOME:/opt/junit/junit.jar:/opt/aspectj/lib/aspectjrt.jar:."
   ```

   On a Windows machine, replace the colons (`:`) in the `CLASSPATH` above by semicolons (`;`). Also, if you're not using bash, set the `CLASSPATH` environment variable from the control panel. (If you don't have JUnit at home, you will need to download that first from `http://www.junit.org`.)

   If you use Eclipse, just include the junit jar file named in the `CLASSPATH` above in your Java Build Path's Libraries path, and work in an AspectJ project. See `http://www.eclipse.org/ajdt/` for more information about using Eclipse with AspectJ.

   To test that everything is working before you start, compile, with `ajc`, the `Fibonacci.java` and `FibonacciTest.java` files, and run `aspectjhw.fib.FobonacciTest`. If you are using the command line on Linux, this is done as follows. First change to the directory immediately above `asjpectjhw` (e.g., `cd ..` if you are currently in that directory), then do:

   ```
   ajc aspectjhw/fib/Fibonacci.java aspectjhw/fib/FibonacciTest.java
   java aspectjhw.fib.FibonacciTest
   ```

   If all goes well, then you should see output like the following.

```
.
Time: 0.015

OK (1 tests)
```

In Eclipse, you should see a green progress bar for the JUnit test if you run FibonacciTest as a JUnit test (or the above output if you run it as an application).

If that doesn't work, carefully review the instructions above, and make sure that everything is installed correctly. If you're still having trouble, ask the course staff for help.

2. (15 points) In this problem you will write a simple development aspect for tracing some Java code. In AspectJ, write an aspect, `aspectjhw.fib.FibTracing`, which you should put in a file `aspectjhw/fib/FibTracing.java`. Define a single named pointcut, `fibpc`, that matches all executions of methods named `fib*` with an argument of type `long`, and two pieces of advice. The first piece of advice should be `before` advice, that prints out "`calling `", the name of the called method (use methods defined on `thisJoinPoint` to get the name — look in API AspectJ defines for this), "`(`", and the argument passed to the call, "`)`", and a newline. You can use `System.out.println` to do this. For example, your code would print "`calling fib(3)`" followed by a newline. The second piece of advice should be `after returning` advice that prints out the result returned by the call, "` == `", the name of the called method (again use `thisJoinPoint` to get the name), "`(`", and the argument passed to the call, "`)`", and a newline. For example, your code would print "`2 == fib(3)`" followed by a newline.

   Now compile your aspect together with `Fibonacci.java` and `FibonacciTest.java`, as follows (on Linux).

   ```
   ajc aspectjhw/fib/FibTracing.java aspectjhw/fib/Fibonacci.java \
       aspectjhw/fib/FibonacciTest.java
   ```

   The tests should pass and the output should also include your tracing information, which starts as follows:

   ```
   calling fib(0)
   0 == fib(0)
   calling fib(1)
   1 == fib(1)
   calling fib(2)
   calling fib(1)
   1 == fib(1)
   calling fib(0)
   0 == fib(0)
   1 == fib(2)
   calling fib(3)
   calling fib(2)
   calling fib(1)
   1 == fib(1)
   calling fib(0)
   0 == fib(0)
   1 == fib(2)
   calling fib(1)
   1 == fib(1)
   2 == fib(3)
   calling fib(8)
   ...
   ```

   Make a printout of your aspect and the testing output, and hand those in for this problem.

3. (15 points) This problem modifies the solution of the previous problem, so you need to make a printout for the previous problem first before continuing.

   In this problem, change the `FibTracing.java` aspect so that it prints a display that indents matching calls and returns by the same amount, and so that a recursive call is nested one space more than the call that originated it. Thus when you compile and run your new tracing output with the Fibonacci test, your output should start like:

```
calling fib(0)
0 == fib(0)
calling fib(1)
1 == fib(1)
calling fib(2)
 calling fib(1)
 1 == fib(1)
 calling fib(0)
 0 == fib(0)
1 == fib(2)
calling fib(3)
 calling fib(2)
  calling fib(1)
  1 == fib(1)
  calling fib(0)
  0 == fib(0)
 1 == fib(2)
 calling fib(1)
 1 == fib(1)
2 == fib(3)
calling fib(8)
...
```

   Be sure to modularize your code so that you don't write the same thing several times.

   Make a printout of your aspect and the testing output, and hand those in for this problem.

4. (15 points) Without changing the `aspectjhw.fib.FibTracing` aspect you wrote in the previous problems or any other existing code in `aspectjhw.fib.Fibonacci` or `aspectjhw.fib.FibonacciTest`, write an aspect, `aspectjhw.fib.BlankLines` that makes the output have a blank line after each successive outermost call to `Fibonacci.fib` returns. (Hint: use `cflowbelow`.) That is the output should start as follows.

```
calling fib(0)
0 == fib(0)

calling fib(1)
1 == fib(1)

calling fib(2)
 calling fib(1)
 1 == fib(1)
 calling fib(0)
 0 == fib(0)
1 == fib(2)

calling fib(3)
 calling fib(2)
```

```
  calling fib(1)
  1 == fib(1)
  calling fib(0)
  0 == fib(0)
 1 == fib(2)
 calling fib(1)
 1 == fib(1)
2 == fib(3)

calling fib(8)
...
```

Make a printout of your aspect and the testing output, and hand those in for this problem.

5. (15 points) Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect, `aspectjhw.fib.FibTiming` that prints out the number of milliseconds taken by each top-level (outermost) call to `Fibonacci.fib` before the blank line printed by the `aspectjhw.fib.BlankLines`. (Hint: use `around` advice and Java's `System.currentTimeMillis()` method.) Your output should thus start as follows.

```
calling fib(0)
0 == fib(0)
Time: 10 ms

calling fib(1)
1 == fib(1)
Time: 0 ms

calling fib(2)
 calling fib(1)
 1 == fib(1)
 calling fib(0)
 0 == fib(0)
1 == fib(2)
Time: 10 ms

calling fib(3)
 calling fib(2)
  calling fib(1)
  1 == fib(1)
  calling fib(0)
  0 == fib(0)
 1 == fib(2)
 calling fib(1)
 1 == fib(1)
2 == fib(3)
Time: 0 ms

calling fib(8)
...
```

Of course the times you actually see will vary depending on the version of Java you use, the operating system, the machine, etc.

Make a printout of your aspect and the testing output, and hand those in for this problem.

6. (15 points) Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect, `aspectjhw.fib.FibMemo` that keeps a table relating arguments and results of `Fibonacci.fib`, and uses these to avoid recomputation previously-computed values for `Fibonacci.fib`. (Hint: use a `java.util.Hashtable` or `HashMap`.)

   When you test, the output should be as follows (modulo the exact times reported, again).

```
calling fib(0)
0 == fib(0)
Time: 16 ms

calling fib(1)
1 == fib(1)
Time: 0 ms

calling fib(2)
1 == fib(2)
Time: 0 ms

calling fib(3)
2 == fib(3)
Time: 0 ms

calling fib(8)
 calling fib(7)
  calling fib(6)
   calling fib(5)
    calling fib(4)
    3 == fib(4)
   5 == fib(5)
  8 == fib(6)
 13 == fib(7)
21 == fib(8)
Time: 0 ms
```

   Make a printout of your aspect and the testing output, and hand those in for this problem.

7. (15 points) You may notice that the test in the above program run noticeably faster than before. We should be able to use the memoized version to compute the Fibonacci function of moderately-sized arguments.

   Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect `aspectjhw.fib.FibLargerTest` that adds another test method, `testLargerArgs` to the class, `aspectjhw.fib.FibonacciTest`, and with body

```
assertTrue(Fibonacci.fib(30) > Fibonacci.fib(20));
assertTrue(Fibonacci.fib(40) > Fibonacci.fib(30));
assertTrue(Fibonacci.fib(50) > Fibonacci.fib(40));
assertTrue(Fibonacci.fib(70) > Fibonacci.fib(50));
```

   Make a printout of your aspect and the testing output, and hand those in for this problem.

8. (15 points) In this and the following several problems we will use the code in the directory `aspectjhw/lambda/` as a base program. This consists of an interface, `Term`, and three classes that implement it: `Variable`, `Application`, and `Lambda`.

   In AspectJ, without changing `Term` or its subtypes, write an aspect `TraceBeta` that prints information about each β-reduction step that results from a call to `Application`'s `reduce1Step`

method. This information must be printed to `System.out`, in a line of the form "`Reducing` e `-->` e'", where e is the application being reduced, and e' is the resulting term. (Hint: Note that all terms support a `toString()` method that can produce the required output. The type of `System.out` supports both a `print` and a `println` method; the `print` method does not put a new line on the output, while the `println` method does.)

For example, with the required aspect, the following program

```
package aspectjhw.lambda;
/** Exercise beta reduction */
public class ExerciseBeta {
    /** Run the exercise */
    public static void main(String[] args) {
        Term t =
            new Application(
                new Lambda("x", new Variable("x")),
                new Variable("y"));
        Term s = t.reduce1Step();
        System.out.println(s.toString());
        t = new Application(
                new Application(
                    new Lambda(
                        "x",
                        new Application(new Variable("x"), new Variable("x"))),
                    new Lambda("z", new Variable("z"))),
                new Variable("q"));
        s = t.reduce1Step();
        s = s.reduce1Step();
        s = s.reduce1Step();
        System.out.println(s.toString());
    }
}
```

would produce the following output.

```
Reducing ((\ x -> x) y) --> y
y
Reducing ((\ x -> (x x)) (\ z -> z)) --> ((\ z -> z) (\ z -> z))
Reducing ((\ z -> z) (\ z -> z)) --> (\ z -> z)
Reducing ((\ z -> z) q) --> q
q
```

Your task is to write an aspect that would produce output like the above in general.

9. (10 points) Answer the question in whichever one of the following two paragraphs applies to you.

   If you solved the previous problem using `around` advice, can you also solve it using only `before` and `after returning` advice? If so do that; if not, describe why it can't be done.

   If you solved the previous problem using only `before` and `after returning` advice, can you also solve it using only `around` advice? If so do that; if not, describe why it can't be done.

10. (15 points) Without changing the code for `Term` or its subtypes, write an aspect, `BigStep` that introduces a method `evaluate()` into the interface `Term` and its three subtypes, `Variable`, `Application`, and `Lambda`. Like the method `reduce1Step()`, the `evaluate` method takes no arguments and returns a `Term`. However, unlike `reduce1Step`, the result returned should be

in β-normal form; that is, if the result of t`.evaluate()` is n, then n`.reduce1Step()` should return a term that `equals` n. (Hint: don't forget that this is a problem about introduction, not about advice.)

A solution should pass the tests in the JUnit test class `aspectjhw.lambda.BigStepTest` that is given for this problem. Note that, in that JUnit test, `assertEquals` compares its two arguments using the `equals` method.

11. (10 points) Without changing the aspects you wrote in the previous problems or any other existing code, write an aspect, `BoundVars` that introduces a method `boundVars()` into the interface `Term` and its three subtypes, `Variable`, `Application`, and `Lambda`. Like the method `freeVars()`, the `boundVars` method takes no arguments and returns a `java.util.Set` of String objects. The set returned should contain exactly the Strings that are the names of variables that occur bound in the term.

    (Hint: use Java's `java.util.HashSet` type, which, like all subtypes of `Set`, supports the methosd `boolean add(Object o)`, `boolean addAll(Collection c)`, note that these mutate the receiver so that it contains `o` or all the elements of `c`, respectively, and they return a boolean status indication that doesn't have to be checked. Also `HashSet` supports the method `boolean contains(Object o)`, which tells if `o` is in the set.)

    Your solution should pass the tests in the JUnit test class `aspectjhw.lambda.BoundVarsTest` we have supplied.

12. (5 points) Suppose that you wanted to modify the behavior of the `reduce1Step()` method for `Lambda` terms so that, instead of simply returning the original term, it checked to see if the η rule of the λ-calculus applied, and did an η-reduction if so. If you wanted to do this without using introductions and without changing the code of `Lambda`, then what kind of advice would you need to use to do that? (Don't write the advice, just say what kind of advice you would need and briefly explain why you must use that kind of advice as opposed to other kinds of advice. For example, `after throwing` is one kind of advice, and `before` is another kind of advice.)

13. (20 points; extra credit) Actually write the aspect code described in the previous probelm to do η-reductions.

14. (40 points; extra credit) Without changing any of the code we supplied or any other aspects you have written so far, change the code to use the applicative order reduction strategy. Currently, the code uses the normal order reduction strategy.

15. (60 points; extra credit) Without changing any of the code we supplied or any other aspects you have written so far, make the code type check terms before applying any reductions. Throw an exception if the term does not type check.

16. (10 points) Does it make sense to use both the `target` and `this` pointcuts in conjunction with an `execution` pointcut? That is, does a pointcut of the form

    ```
    execution(* f()) && target(x) && this(y)
    ```

    make sense? Briefly explain your answer.

17. (10 points) Can one use AspectJ to enforce the following programming rule?

    > Code in package `b` may not call code in package `a`?

    If so, show how this would be done in AspectJ; if not, then explain why this is impossible.

18. (5 points) What pointcuts in AspectJ can "bind" formals declared in advice or a pointcut declaration?

19. Suppose AspectJ were extended with a primitive pointcut `arrayfetch`, that takes a field pattern as an argument. For example, the pointcut

    ```
    arrayfetch(Term[] Application.exps)
    ```

    describes the joinpoints in the class `Application` which occur when the array `exps`, of type `Term[]`, has elements fetched from it. To simplify the discussion below, let us suppose that, in general, this pointcut has the form `arrayfetch(`JT`[] `TP`.`$IP`)`, where JT is a Java type (the type of the array's elements), TP is a type pattern that describes the types where the field is to be found, and IP is an identifier pattern that describes the names of the array fields that are being observed.

    (a) (5 points) If a joinpoint of the form `arrayfetch(`JT`[] `TP`.`$IP`)` is used in `around` advice, what type constraints should there be on the return type of a `proceed()` expression in the body of the `around` advice?

    (b) (5 points) If a joinpoint of the form `arrayfetch(`JT`[ ] `TP`.`$IP`)` is used in `after returning` advice, what type constraints should there be on the return type of a formal parameter that is declared for the return value in such advice?